

Navigate The Mars Rover

TECHNICAL REPORT



July 24, 2020

Microsoft Codess’20

TEAM:trouble\_shooter

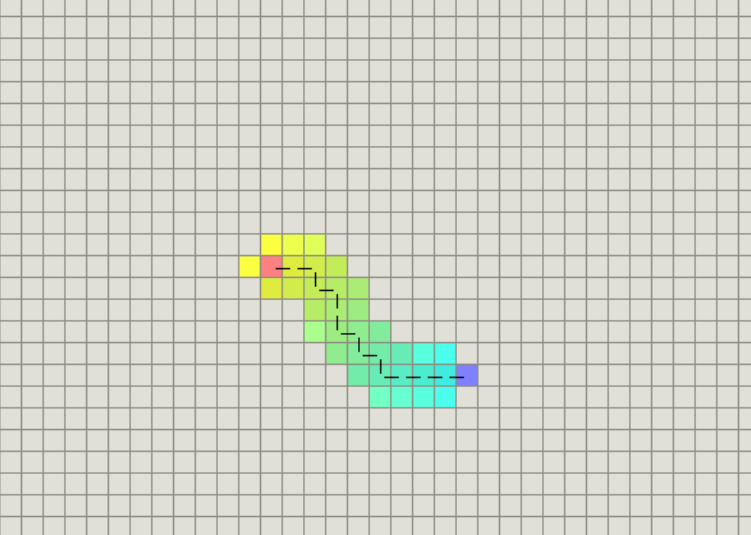


Table of Contents

[Introduction to project 3](#_Toc46482285)

[Server 3](#_Toc46482286)

[Browser 3](#_Toc46482287)

[Basic Usage 4](#_Toc46482288)

[Advanced Usage 6](#_Toc46482289)

[Development 7](#_Toc46482290)

[Technologies 8](#_Toc46482291)

[Conclusion 8](#_Toc46482292)

[References 9](#_Toc46482293)

[Acknowledgement 9](#_Toc46482294)

# Introduction to project

The aim of this project is to provide a path-finding library that can be easily incorporated into web games. It runs on Node.js .

It comes along with a browser based **web application** to show how the algorithms execute the Mars Curiosity Rover find the shortest path between two points while avoiding obstacles on the way. .

***Note*** : This project provides path-finding algorithms for 2D space.

# Server

If you want to use it in Node.js, you may install it via npm.

npm install pathfinding

Then, in your program:

var PF = require('pathfinding');

# Browser

If you have browser installed then you can install it with the following command:

Git link

By default browser will install pathfinding under the browser components folder, so to include it in your page do something like:

<script type = “txt/javascript”

src ="script.js"></script>

# Basic Usage

To build a grid-map of width and height 39 pixels, the code is as follows:

<div id="grid">

        <%for(let i=0;i<23;i++){%>

        <div class="row">

            <%for(let j=0;j<48;j++){%>

            <div class="cell" rid=<%=i%> cid=<%=j%>>

            </div>

            <%}%>

    </div>

By default, all the cells in the grid will be able to be walked through. First we will click on the cell where we want our source to be placed, then we will click on the cell where we want our destination to be placed. After placing the source and destination we will click on the cell where we want our obstacles to be placed in between the path.

For example, to set the cell at (0, 0) to be un-walkable, where 0 is the x coordinate (from left to right), and 0 is the y coordinate (from up to down). Similarly we will select our destination and obstacles. The code for the same is as follows:

for (let i = 0; i < allCells.length; i++) {

    allCells[i].addEventListener("click", function () {

        let rid = Number(allCells[i].getAttribute("rid"));

        let cid = Number(allCells[i].getAttribute("cid"));

        if (!src) {

            allCells[i].style.backgroundColor = "green"

            src = [rid, cid];

            console.log(src);

        } else if (!dest) {

            allCells[i].style.backgroundColor = "red"

            dest = [rid, cid];

            console.log(dest);

        } else {

            allCells[i].style.backgroundColor = "gray"

            obstacles.push([rid, cid]);

        }

    })

}

We pass the grid matrix from grid.ejs and script.js to the server file. It will initiate all the cells in the grid with the same walkability indicated by the matrix. When we select our source, destination and obstacles, 1 will be our source, 2 will be our destination , 0 means that the cell is walkable and -1 means obstacles are there making them unwalkable.

int[][] matrix = [

[1, 0, 0, -1, 0],

[-1, 0, 0, 0, -1],

[0, 0, -1, 0, 2],

];

Currently there are 4 path-finders bundled in this library, namely:

* AStar Finder \*
* Best First Finder
* Breadth First Finder \*
* Dijkstra Finder\*

In these algorithms, we have two option which are allow diagonal and don’t allow diagonal.

Also, Note that only the finders with trailing asterisks are guaranteed to find the shortest path.

To build a path-finder, say, AStarFinder we will create an object in the mainfun.java file and the call the constructors and pass the arguments:

astaralgo a=new astaralgo(matrix,source,destination,typeheuristic,

isDiagonal);

a.astarAlgo();

To find a path from (1, 2) to (4, 2), (Note: both the start point and end point should be walkable), we are passing the arguments in the constructor of the object as required by the algorithm.

Source and destination will be an array of coordinates.

For the matrix defined previously, the path will be :

[ [ 1, 2 ], [ 1, 1 ], [ 2, 1 ], [ 3, 1 ], [ 3, 2 ], [ 4, 2 ] ]

* We can change the position of source ,destination and obstacles as per our requirement .Even we can use the same grid multiple times because calling of the function takes place from the main function and not the algorithm directly.
* We have optimised the algorithm by passing the source and destination arrays as arguments while calling the function because the defined variables in the class of required algorithm will directly access the positions of source and destination in O(1) time complexity otherwise it would have taken O(n^2) time to find the source and destination in the grid (matrix).
* We have used min priority queue based implementation in the algorithm to find the optimised path. The complexity for min priority queue in this grid is O(n\*log(n)).

# Advanced Usage

When instantiating path-finders, you may pass in additional parameters to indicate which specific strategies to use.

For all path-finders, you may indicate whether diagonal movement is allowed. The default value is true, which means that the path can only orthogonally as well as diagonally.

In order to disable diagonal movement:

We will click on Don’t Allow Diagonal option while selecting the algorithm.

For AStarFinder, BestFirstFinder, you may indicate which heuristic function to use.

The predefined heuristics are :

* Manhattan
* Euclidean
* Chebyshev (Diagonal)

To use the Chebyshev (diagonal) heuristic, we will select the heuristic while selecting the algorithm.

To build a BestFirstFinder with diagonal movement allowed and a custom heuristic function, we will choose the Best First Search Algorithm, Manhattan Heuristic and Allow Diagonal Option.

# Development

Layout:

.

|-- lib # browser distribution

|-- src # source code (algorithms only)

|-- test # test scripts

|-- utils # build scripts

|-- benchmark # benchmarks

`-- visual # visualization

Make sure you have node.js installed, then use npm to install the dependencies:

npm install -d

The build system uses gulp, so make sure you have it installed:

npm install -d -g gulp

To build the browser distribution:

gulp compile

To run the tests (algorithms only, not including the visualization) with [mocha](http://mochajs.org/) and [should.js](https://github.com/visionmedia/should.js) First install mocha:

npm install -d -g mocha

Then run the tests:

gulp test

To run the benchmarks:

gulp bench

Or if you are feeling lazy, the default gulp task does everything(except running the benchmarks):

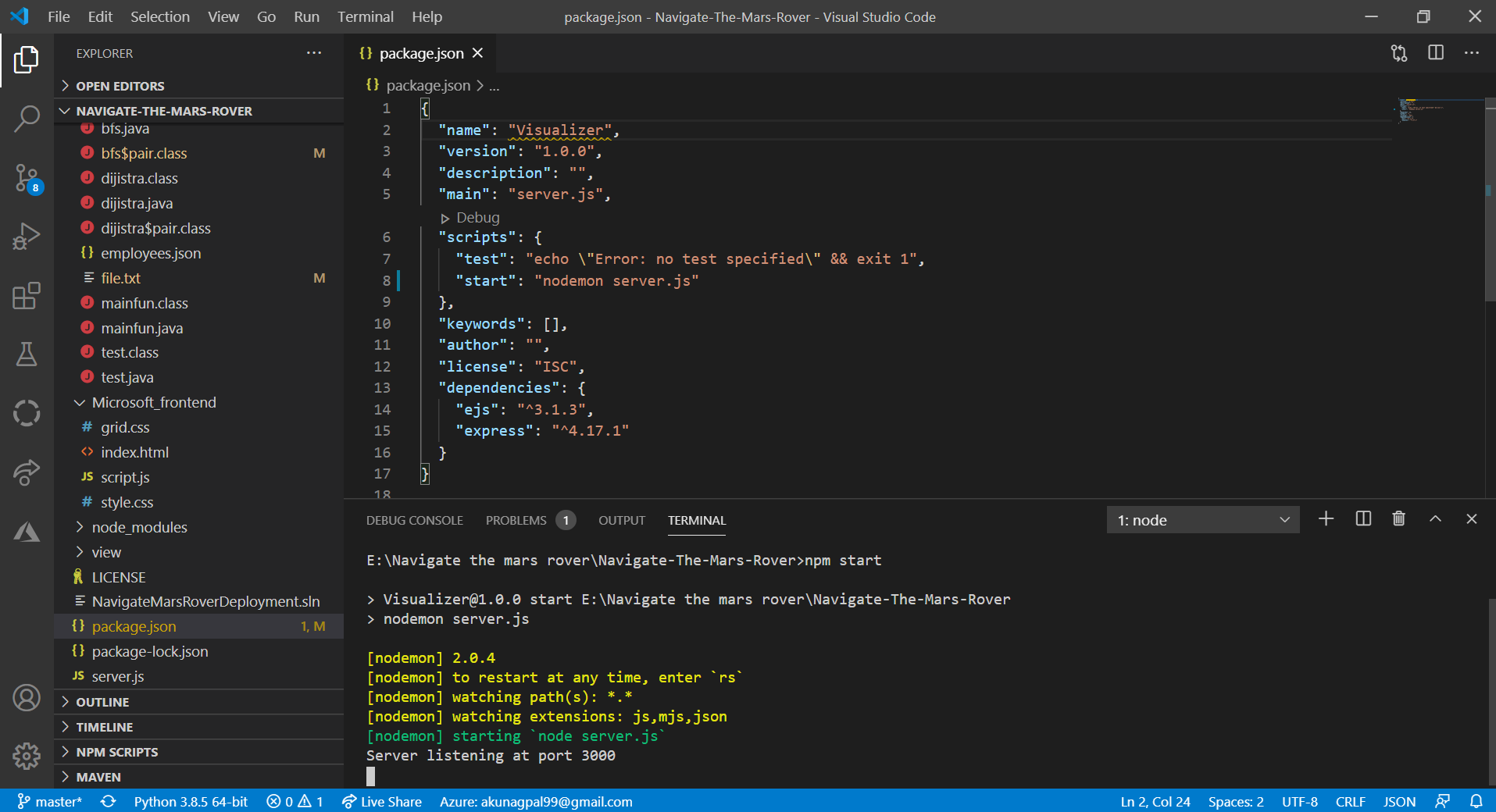
Gulp

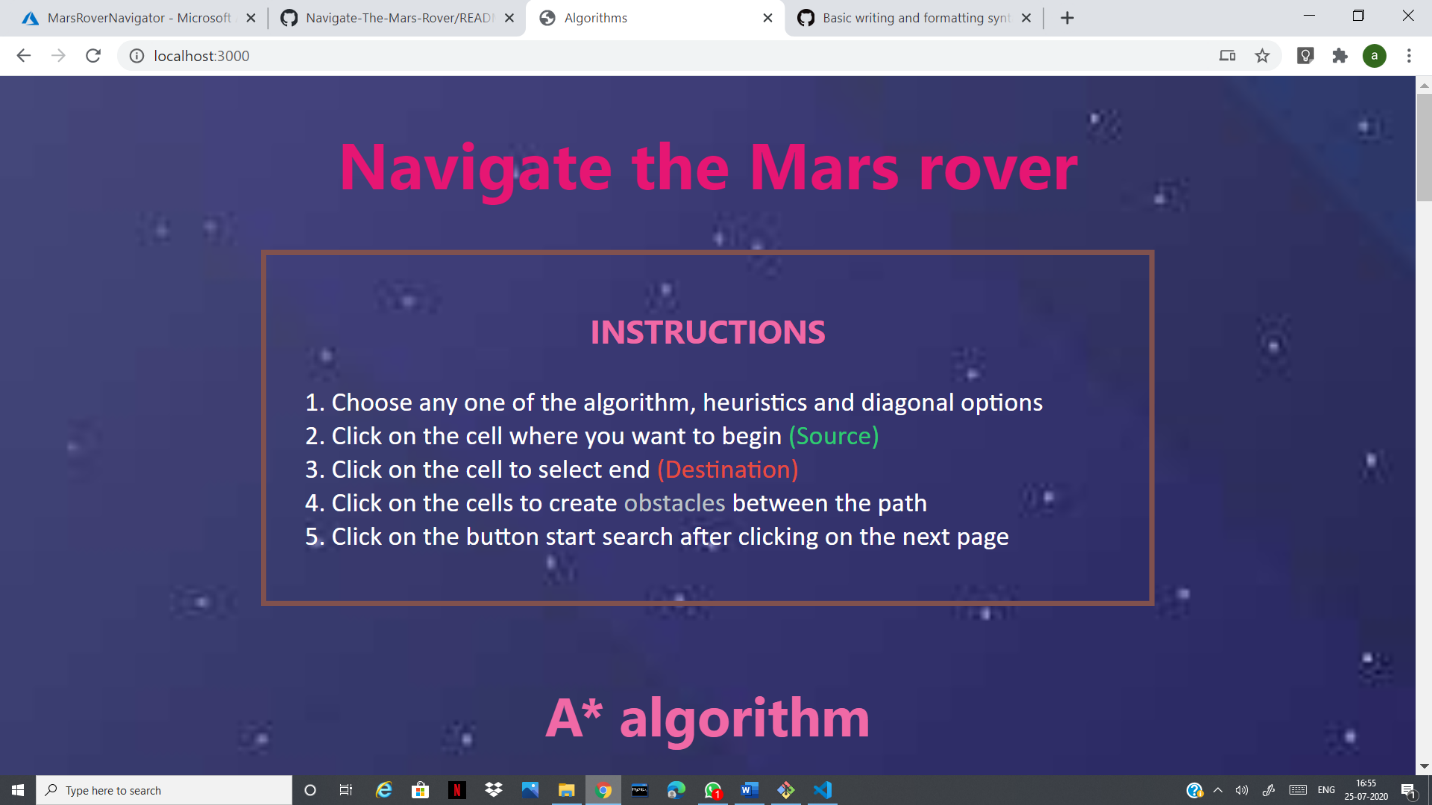
# Technologies

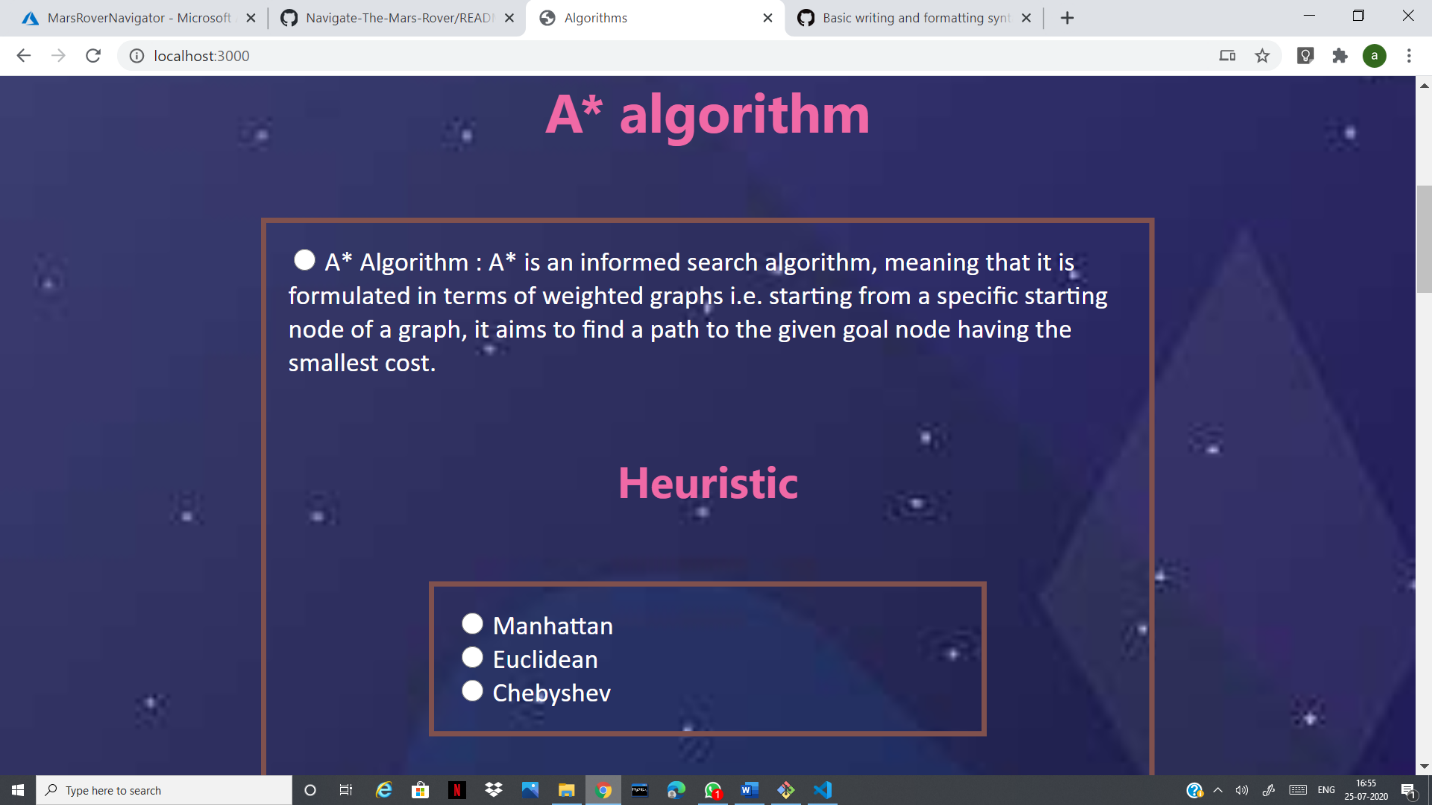
Listing down the following technologies:

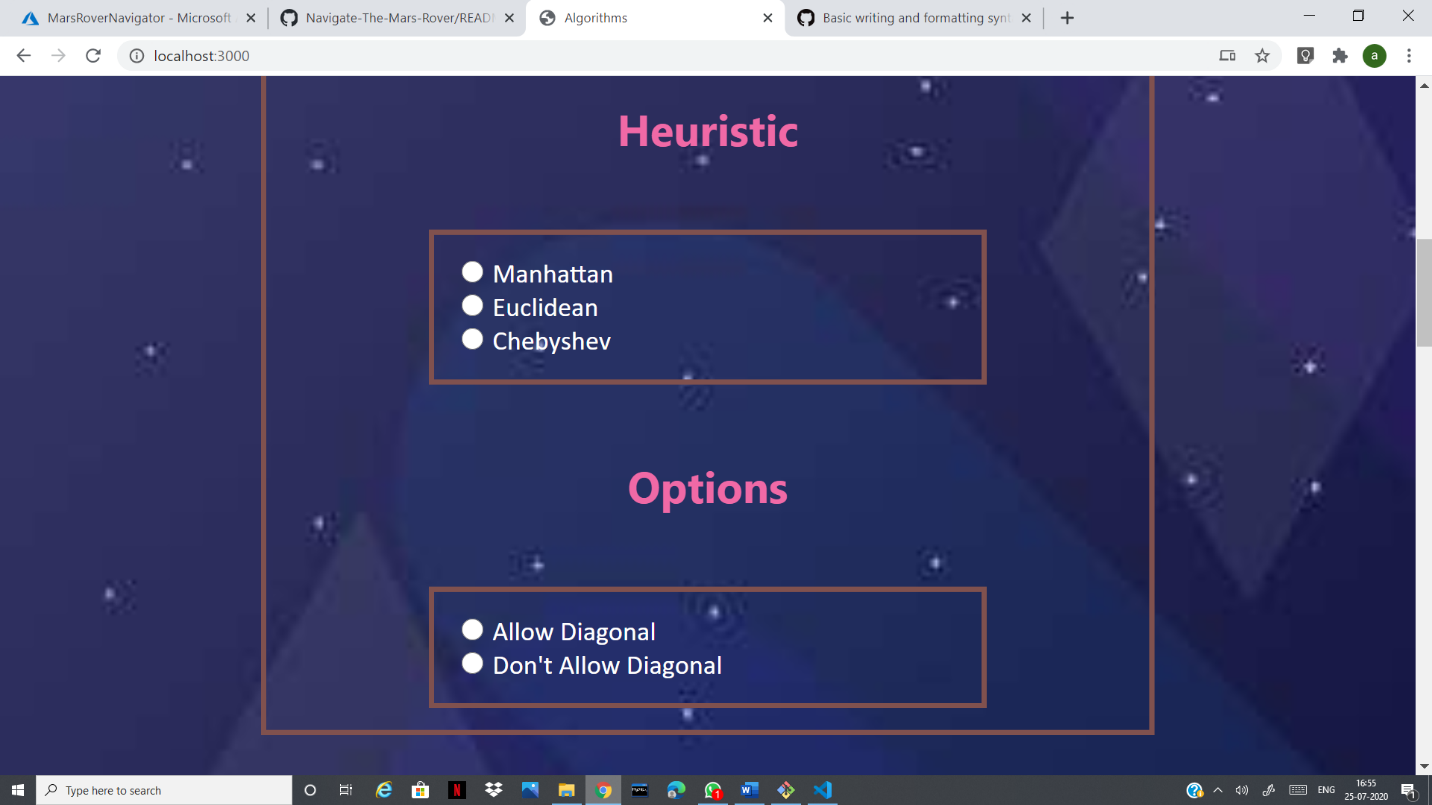
* Hyper Text Markup Language (HTML)
* Cascading Style Sheets (CSS)
* JavaScript (JS)
* JavaScript Object Notation (Json)
* Embedded JavaScript (EJS)
* Java Programming
* Node Package Manager (NPM)
* Node JS

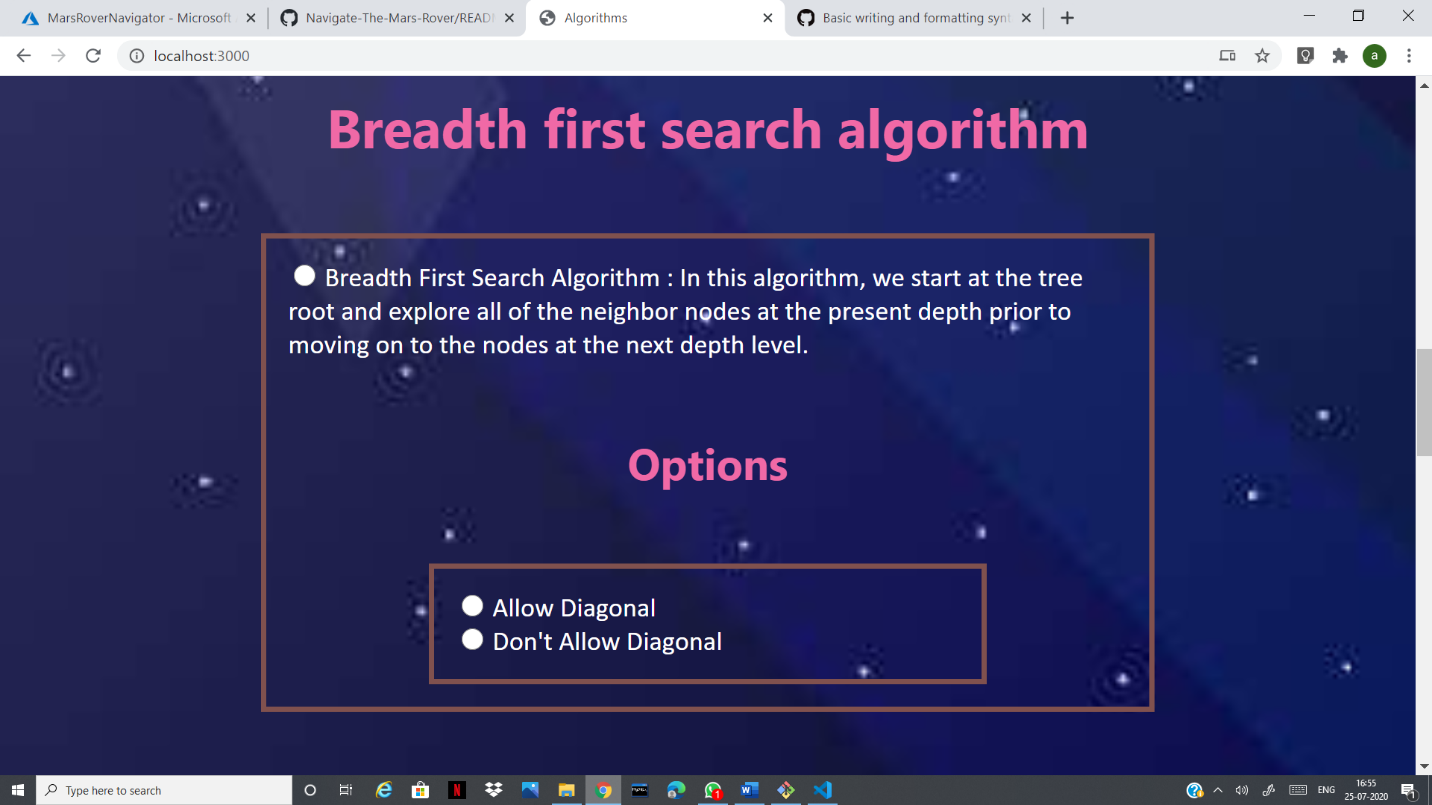
# Result

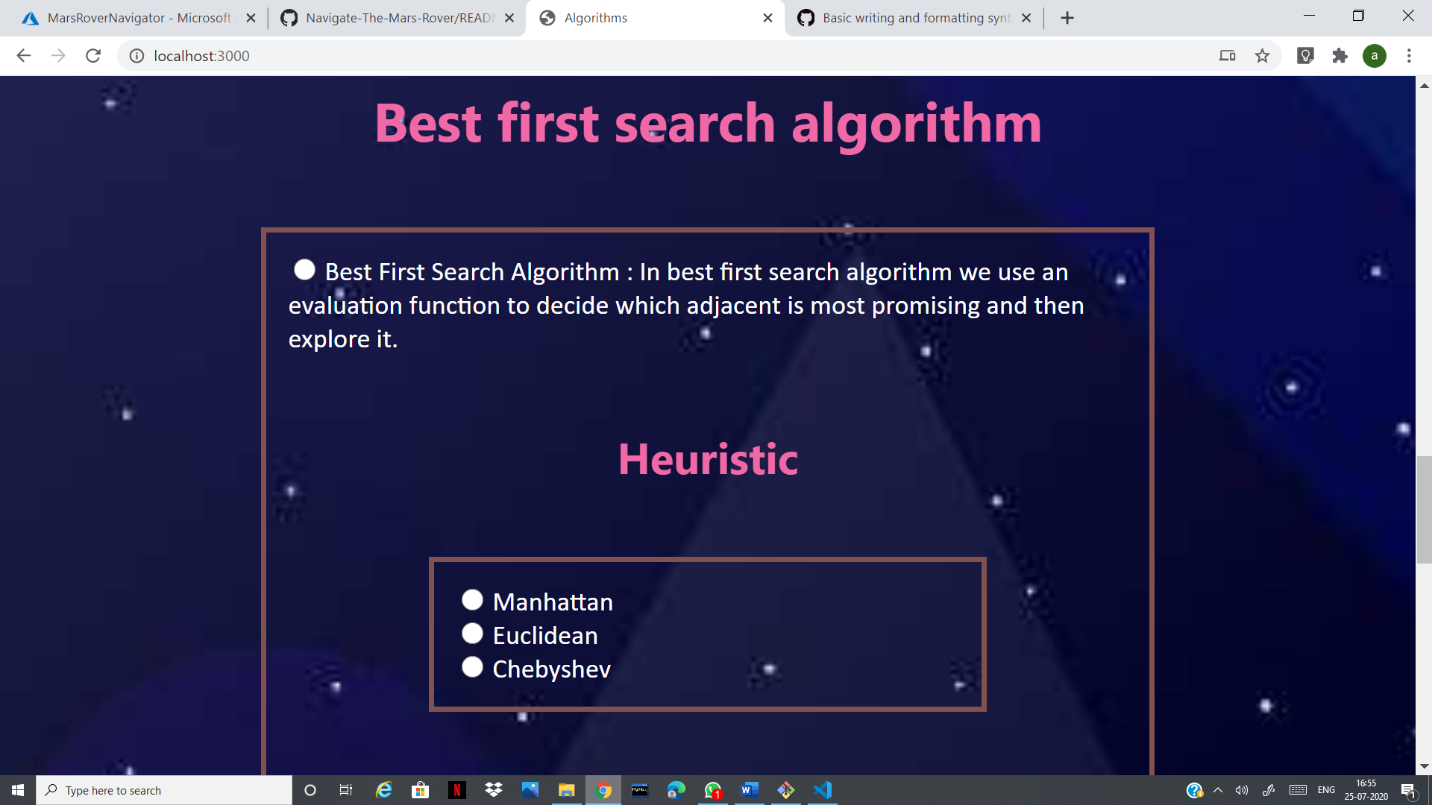


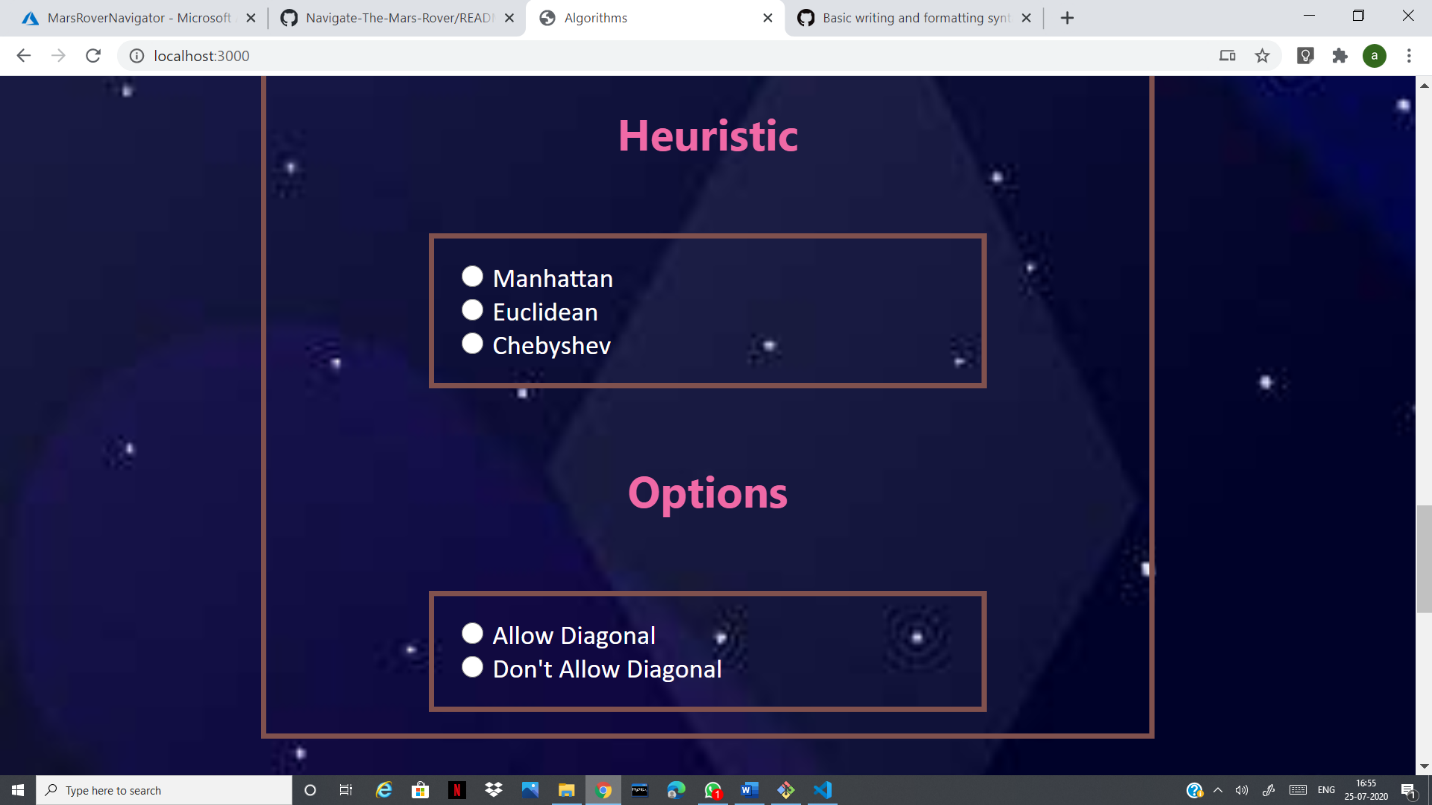


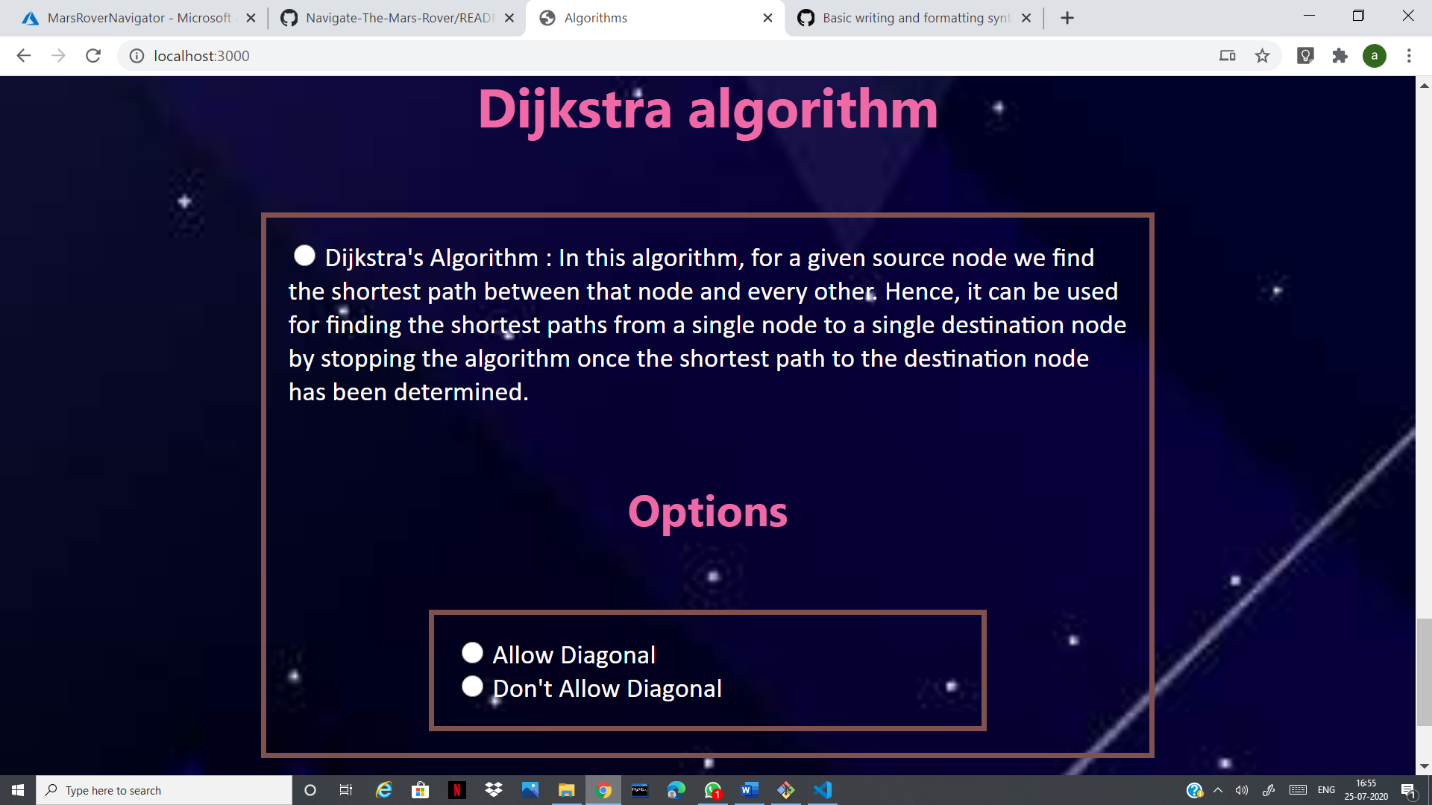


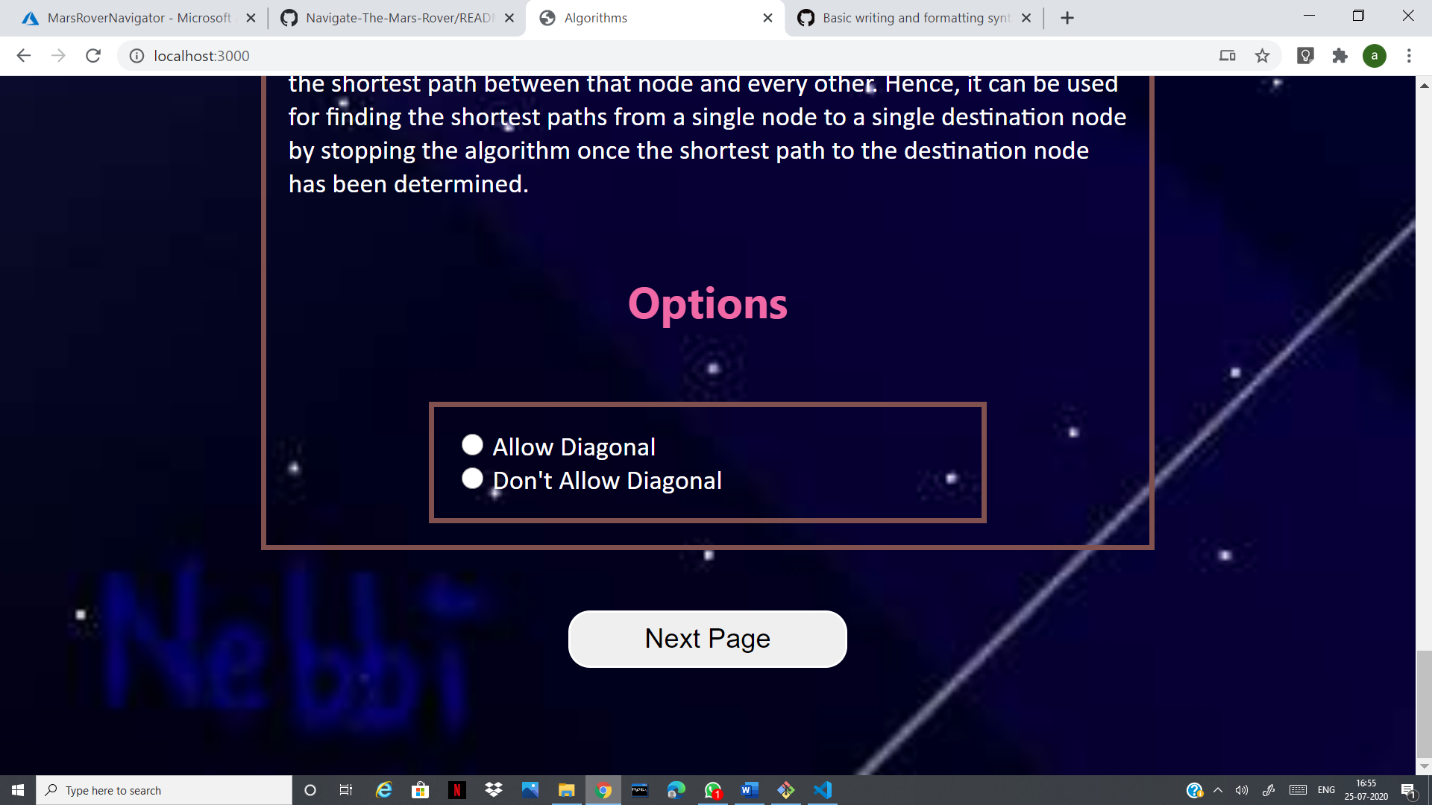


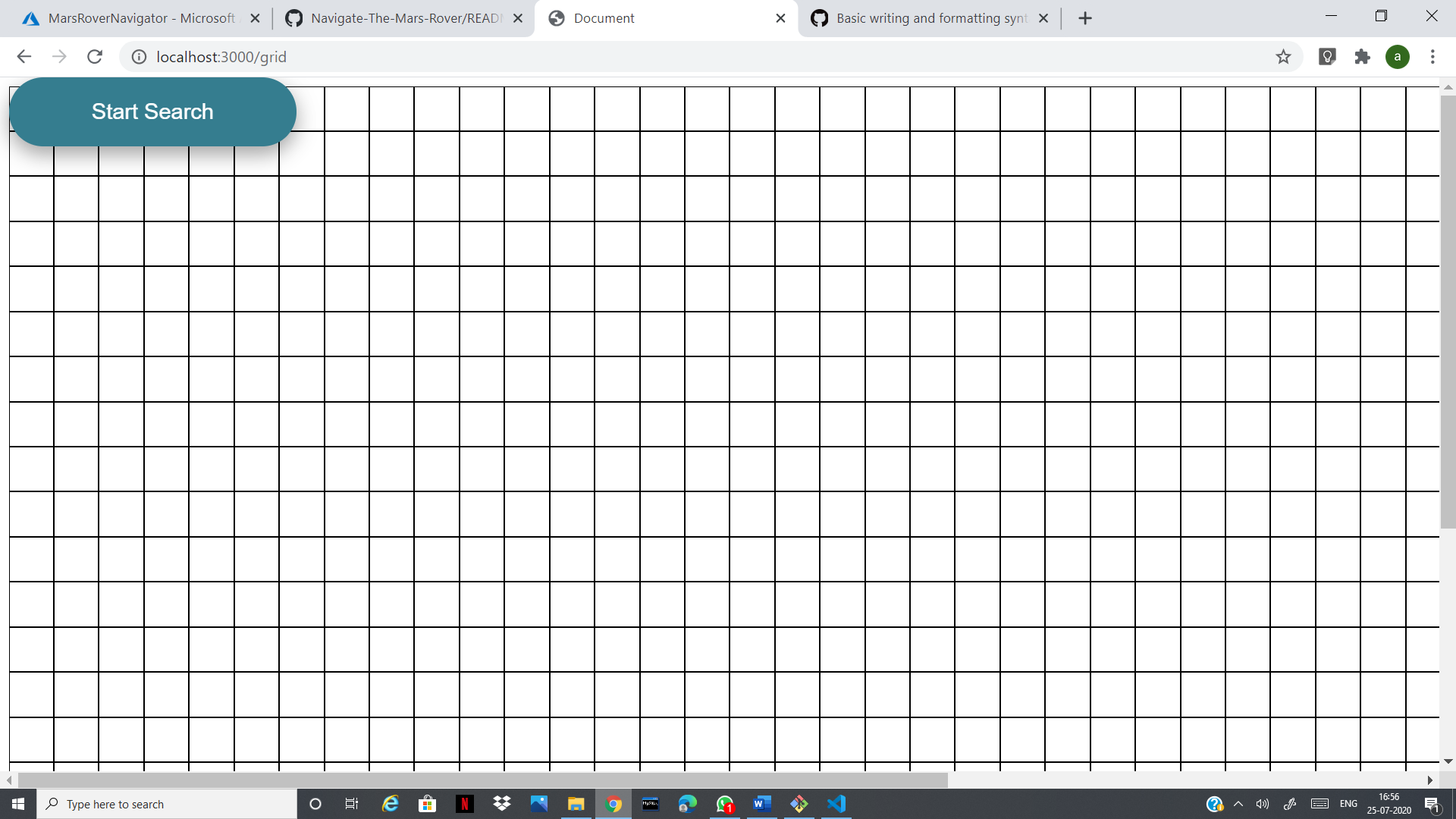


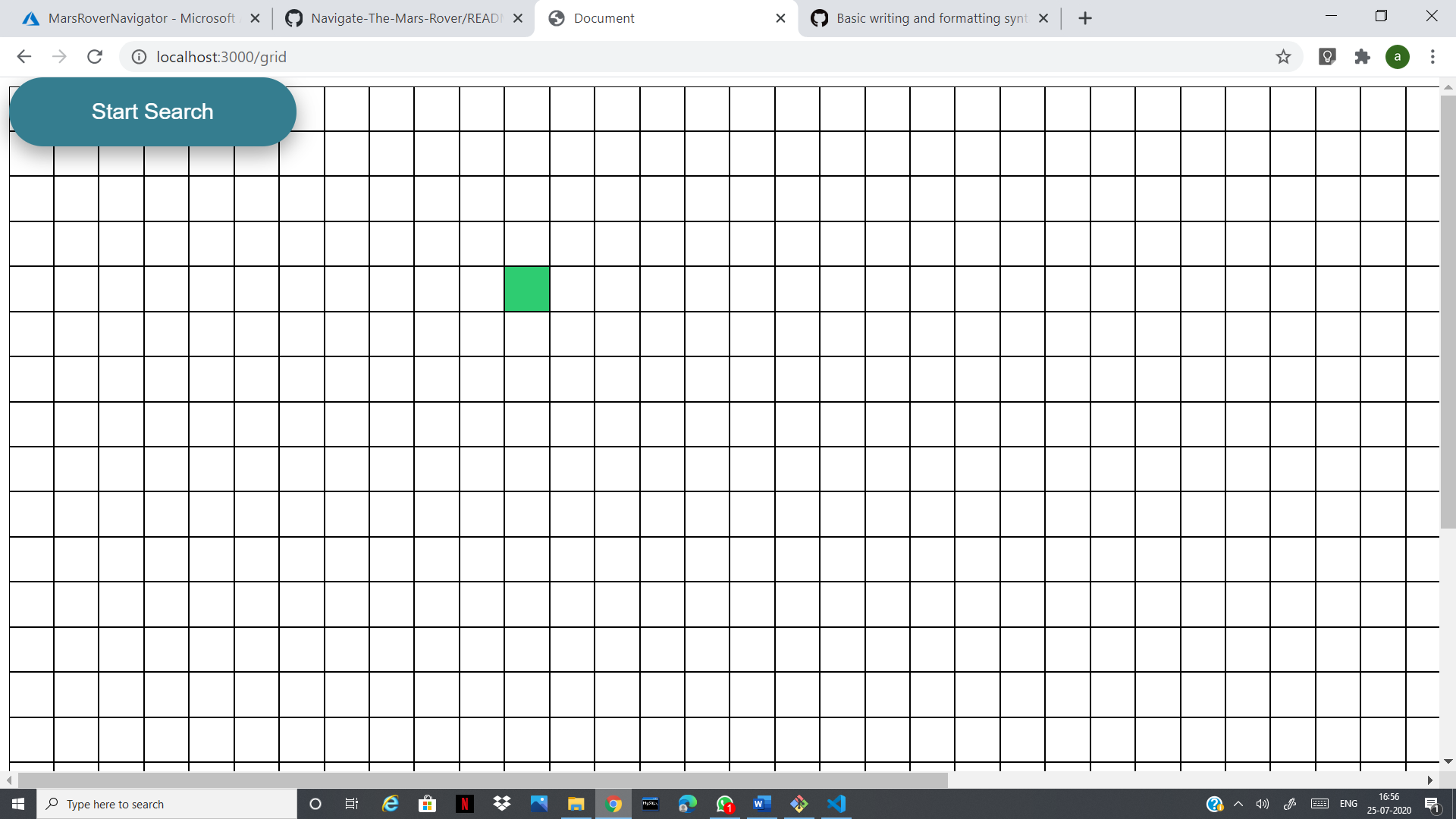


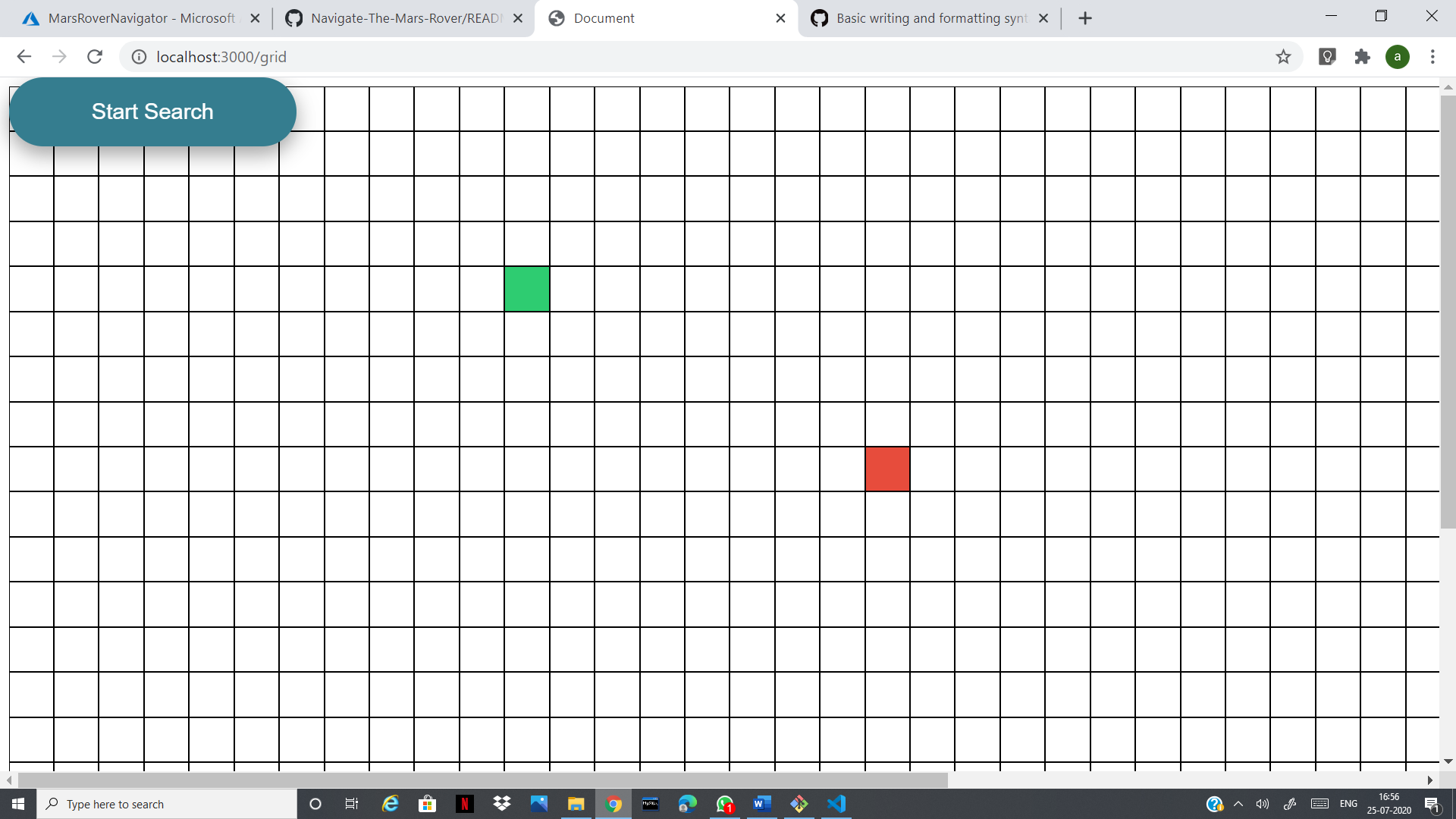


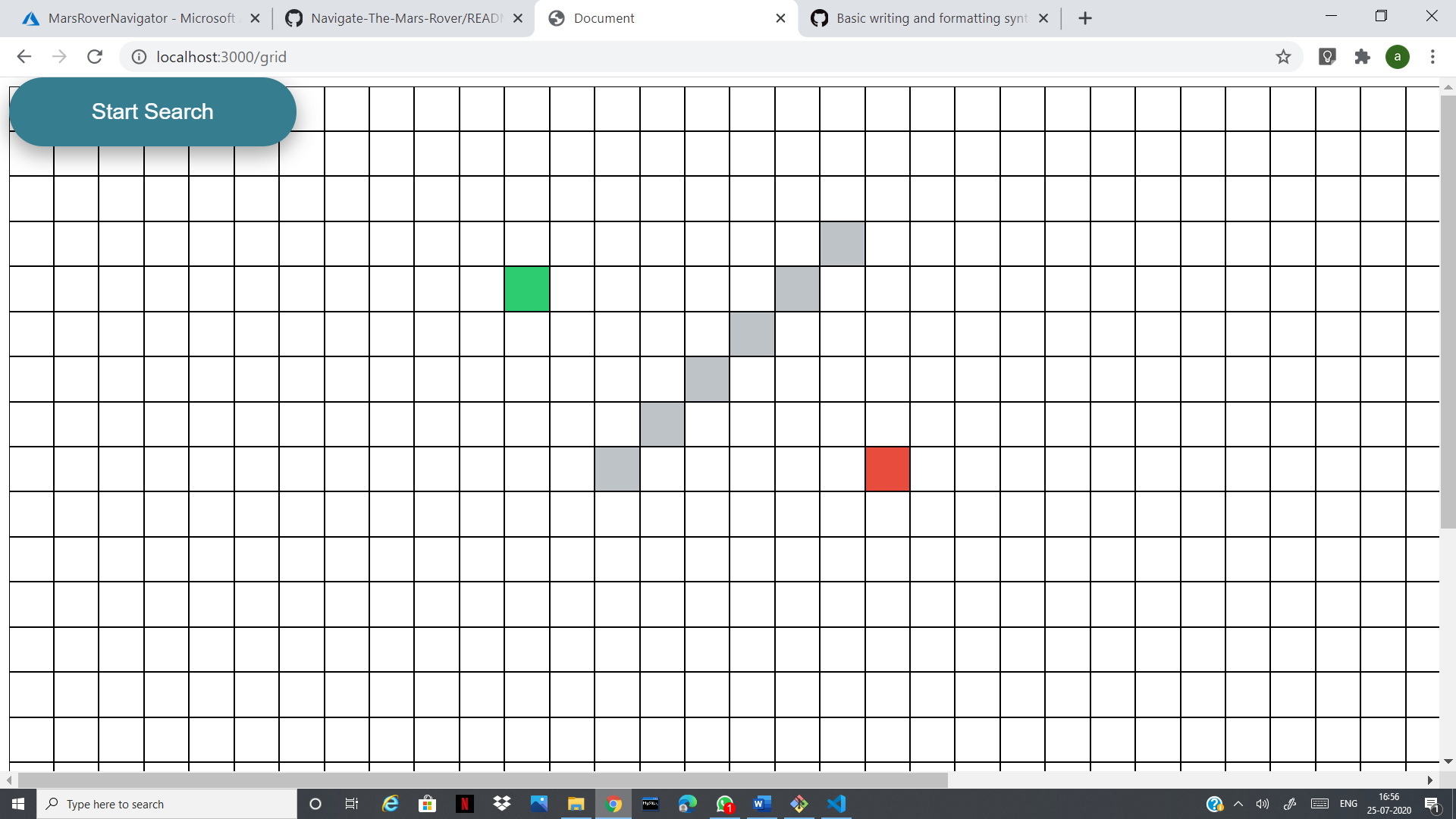


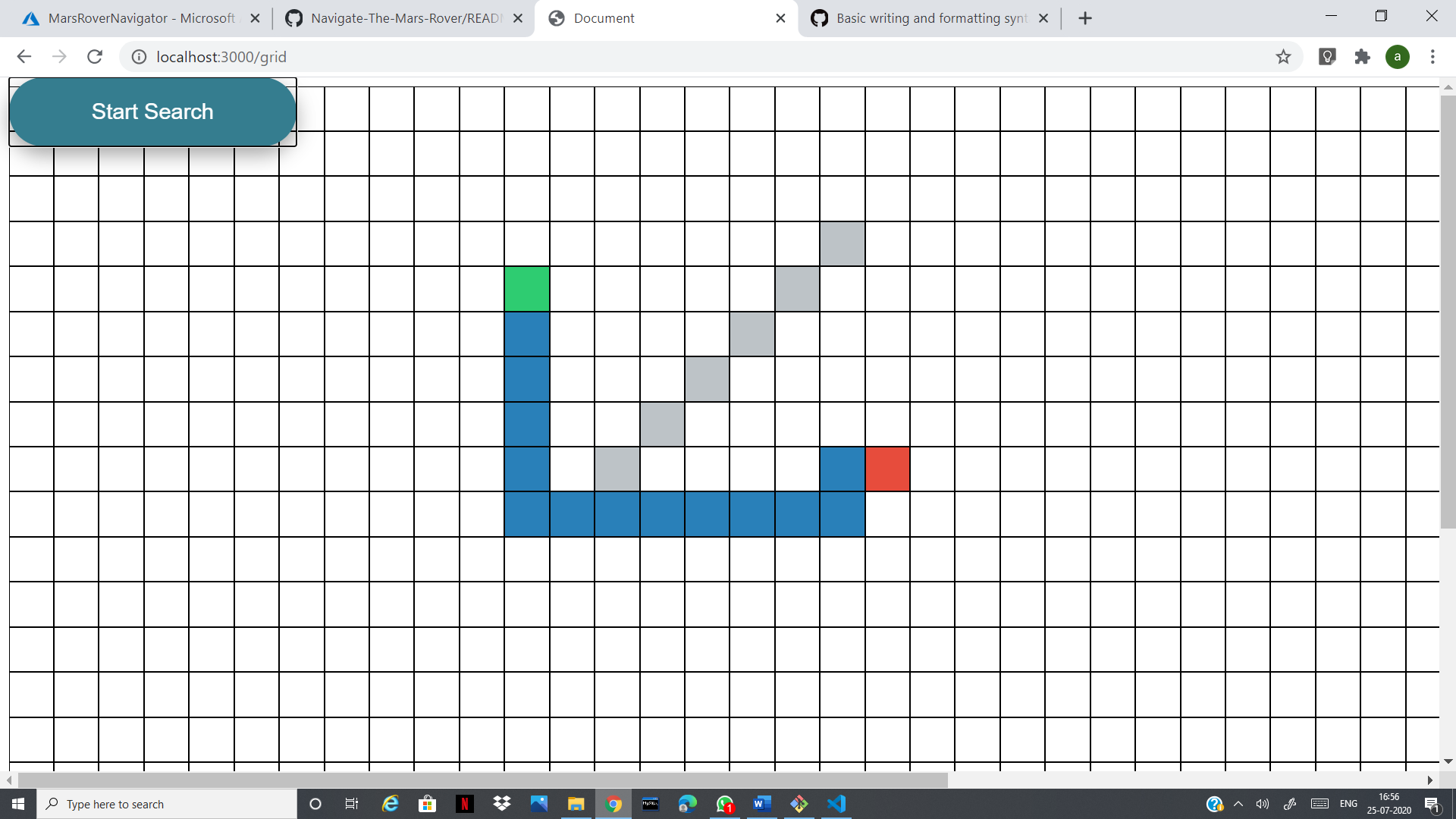












# Conclusion

The web application *Navigate The Mars Rover* would be finding the shortest path between source and destination without having collision with number of obstacles present in between.

The user can use **A Star Finder**, **Best First Search**, **Breadth First Search** or **Dijkstra** path finding algorithms. Using this user can also compare the efficiency of different algorithms mentioned above. The user can have any type of heuristic as per their choice and allowing diagonal movement or not is solely the choice of the user.

The time complexity of A Star Finder algorithm is *O(log h\*(x))* where, *h* is optimal heuristic cost and *x* is the cost from *x* to the destination, Best First Search is *O(n log n)* where *n* is number of nodes in between, Breadth First Search and Dijkstra is *O(V+E)* where *V* is the number of nodes traversed and *E* is the number of edges between source and destination.

A\* algorithm is built using Dijkstra algorithm so it is more efficient because it used heuristic approach. This fact user can easily make out while finding the shortest path. Finally, we computed shortest path using all the above mentioned algorithms.

# References

* CORMEN Book
* MIT OCW
* W3S schools
* Udemy Course - The web developer bootcamp
* Mozilla Developer Network(MDN) Web Docs

# Acknowledgement

We would like to express our deepest gratitude to the program Engage 2020 for providing us the opportunity to work for this project.

We would like to specially thank our mentors for staying in touch and through 1:1 sessions widening our horizon of thinking that we had used in our project. We would also like to acknowledge each other for assembling different segments for this project and completing it successfully. Last but not the least we would convey our joy and thanks to god almighty for his never ending grace in this pandemic situation.

Regards

Team : trouble\_shooters